Solar Physics
DOI: 10.1007/●●●●●-●●●-●●●-●●●●-●

A Micro-Service-Based Framework for Multi-Level Testing of Cyber-Physical Systems

Iñigo Aldalur[1] · Aitor Arrieta[1] · Aitor Agirre[2] · Goiuria Sagardui[1] · Maite Arratibel[3]

Abstract In the last years, the use of microservice architectures is spreading in Cyber-Physical Systems (CPSs) and Internet of Things (IoT) domains. CPSs are systems that integrate digital cyber computations with physical processes. The development of software for CPSs demands a constant maintenance to support new requirements, bug fixes, and deal with hardware obsolescence. The key in this process is code testing and more if the code is fragmented during the development of CPSs. It is important to remark that this process is challenging and time-consuming. In this paper, we report on the experience of instantiating of the microservice-based architecture for DevOps of CPSs to test elevator dispatching algorithms across different test levels (i.e., SiL, HiL and Operation). Such an architecture allows for a continuous deployment, monitoring and validation of CPSs. By integrating the approach with a real industrial case study, we demonstrate that our approach reduces significantly the time needed in the testing process and consequently, reduces the economic cost of the entire process.

Keywords: Cyber-Physical Systems, Elevators, Testing

✉ Iñigo Aldalur
ialdalur@mondragon.edu

✉ Aitor Arrieta
aarrieta@mondragon.edu

✉ Aitor Agirre
aagirre@ikerlan.es

✉ Goiuria Sagardui
gsagardui@mondragon.edu

✉ Maite Arratibel
marratibel@orona-group.com

[1] Mondragon Unibertsitatea, Spain

[2] Ikerlan, Spain

[3] Orona, Spain

1. Introduction

Cyber-Physical Systems (CPS) are defined as an integration of digital cyber computations with physical processes (Lee and Seshia, 2017). CPSs involve different aspects, such as distributed computing, communication, and automatic control, and they are applied in many domains such as the automotive, railway, healthcare, entertainment, elevation and city infrastructure (Stankovic, 2016). The software maintenance process of CPSs is constant, and can last up to 30 years. Maintenance activities in such a context encompass, but are not limited to correction of bugs, inclusion of new functionalities, dealing with hardware obsolescence and new legislative changes (Ayerdi *et al.*, 2020b). This is the example of our industrial case study, i.e., the elevator dispatching algorithm. In such a context, releasing a new software version takes between 6 and 12 months time. As compared to other type of software systems (e.g., web applications, mobile apps), this time is long.

To a large extent, this period is long due to the high effort required by software engineers when testing the software. When testing CPSs, several test levels exist (Ayerdi *et al.*, 2020b), namely, the Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), Hardware-in-the-Loop (HiL) and physical testing. Usually, there are interoperability issues when trying to execute test cases at MiL and SiL at higher test levels (e.g., HiL). For instance, the tools at MiL, SiL and HiL may vary, and therefore, significant manual effort is spent to adapt test cases from a test level to another. Something similar happens with test oracles, which require manual effort to be adapted from a test level to another. Despite several interoperability standards having emerged (e.g., ASAM-XiL), these are only valid for some types of CPS simulators.

To solve this issue, in this paper, we propose a multi-level testing approach for CPSs. Multi-level testing is the use of different test cases that can be reused across test levels and test platforms (Perez and Kaiser, 2009). When designing multi-level testing, the aim will be to obtain the uppermost possible level of independence from the platform within the test cases. To this end, we propose a solution that is based on microservices, which allows for re-using test cases and test oracles across different test levels, i.e., SiL, HiL and Operation. We develop a prototype of our approach and apply it in an industrial case study from the elevation domain, which is provided by Orona, one of the largest elevator companies in Europe. In the paper, we report on the experiences of applying this approach in the industrial case study. Specifically, we make the following contributions:

- We propose a microservice-based framework that allows for executing test cases at different test levels (SiL, HiL and Operation), and which is integrated with a DevOps ecosystem for CPSs.
- We instantiate the approach in a real industrial case study encompassing the dispatching algorithm of elevators, provided by Orona, one of the largest elevator companies in Europe.
- We carry out an evaluation (both qualitative and quantitative) of the adoption of the approach by our industrial partner.

The rest of the paper is structured as follows. Section 2 analyses the problem we want to solve, its causes and its consequences. Section 3 discusses the related work in order to give the reader an idea of what has been done in this area. Section 4 presents the case study of this contribution. Section 5 describes the defined concepts that verify and validate the multi-level testing plan for CPSs. Sections 6, 7 and 8 describe the developed process for testing our case study based on microservices. Section 9 evaluates our approach, by means of an industrial case study, demonstrating that our process saves considerably the time needed in the multi-level testing process, and consequently it saves money for the company that has implemented the process. Lastly, section 10 concludes the paper and describes the next steps for the future.

2. Problem analysis

2.1. Problem

There has been a rapid development of CPSs. Such systems operate in uncertain environments, and high Quality of Service (QoS) has become a mandatory prerequisite in industry. The design of CPSs is inherently challenging and time-consuming due to their competing requirements regarding affordability, performance, safety, security, sustainability, etc. (Masin *et al.*, 2017). Moreover, not only makes the lack of test automation in CPS engineering testing activities time-consuming, but also they are time-consuming (Hu *et al.*, 2020). Carrying out appropriate mechanisms that handle faults in run-time demands a plenty of know-how, and it is frequently error-prone (Hu *et al.*, 2020).

2.2. Causes

The first cause is the lack of reusability and interoperability between test artifacts among the different test levels. For example, one of robotics software main obstacles in industry and academia is the lack of interoperability and reuse of software libraries (Bruyninckx, 2008). Being one of the main drawbacks in the robotics community, it is common to see works that avoid the lack of code reuse and develop methods that facilitate the integration of different components and interoperability (González-Nalda *et al.*, 2017). Furthermore, taking into account that CPS experiments are very time-consuming and economically expensive, some works have developed methods that reuse components for experiments. These components must be interoperable with each other (Neema *et al.*, 2019). The second cause is the strong focus and dependability on certain domain, for example, automotive domain. The weakness in this domain is the dependability with the variety of complex and heterogeneous architectures because they are exposed to several vulnerabilities (Veledar, Damjanovic-Behrendt, and Macher, 2019). "Dependability represents the ability to deliver service that can justifiably be trusted" (Avizienis *et al.*, 2004). CPS's goal is assuring dependability (Armengaud *et al.*, 2018). However, dependability is not innate in CPSs. At the beginning of the design process, it is mandatory to provide feedback to the

system to reduce the risk of late discovery of dependability (Miele, 2014). System processes must be conducted adequately with no interruptions, which makes assuring CPSs' dependability a challenging activity to complete satisfactorily (Sanislav, Mois, and Miclea, 2016).

The third cause is that in CPS, design-time methods are different from operation time methods. Different designers usually develop these methods. When designers are expert in a field, and they focus on the main aspects of their expertise, they neglect or avoid considering other important aspects. This might cause that some requirements are not completely fulfilled. Moreover, it is common a lack of communication between different designers (Sundharam *et al.*, 2018). Design-time methods are implemented the more efficient and stable as possible. Operation-time methods, however, are concerned with integrating many different applications onto the same target platform (Lampke *et al.*, 2015). The last cause is the lack of automatic oracles, PS domain test oracles are likely to be manual in operation-time which is the main cost in testing (Matinnejad *et al.*, 2019). Unfortunately, the decision of whether a test case has succeeded or not is manual (Gartziandia *et al.*, 2021). It is common in CPS the manual judge of the results whose success depends on the expertise and availability of the expert, which is not practical (He *et al.*, 2019). The main drawback is that most testers are experts in one domain (automotive or space engineering), but they are not necessarily experts in software engineering (Shin *et al.*, 2021).

2.3. Consequences

The main consequence of the lack of test automation in CPS engineering multi-level testing activities is that they are accomplished manually. Hardware-in-the-loop (HiL) testing manipulates hardware, and they need a huge quantity of time to be developed and also to be executed. It is extremely important that these test cases are correctly performed not to damage the hardware and to be executed in a period of time (Shin *et al.*, 2021). The creation of these test cases for the experiments is time-consuming and error-prone, needing a significant manual exertion before they can be utilized as HiL experiments (Wang *et al.*, 2015). Another consequence of the manual creation of test cases is the elevated economic cost for companies. The more time a tester needs to develop test cases, the higher the economic cost for the company. Additionally, solving testing errors prolongs the time needed to create correct test cases. Even if it is complicated evaluating accurately the cost of implementing and operating CPSs, models have been proposed to calculate this cost (Bataev and Aleksandrova, 2020).

3. Related work

In the last few years, verification of CPSs has centered a large attention from the software engineering research community Sadri-Moshkenani, Bradley, and Rothermel (2022). Different perspectives have been tackled. One of the most studied perspectives is test generation. Since applying model-based testing in such systems is difficult due to the challenges of capturing the physical behavior Briand *et al.* (2016), many approaches advocated for search-based test

generation techniques for cost-effectively generating test cases Matinnejad *et al.* (2019); Menghi *et al.* (2020); Arrieta *et al.* (2017c,b); Khatiri, Panichella, and Tonella (2023); Humeniuk, Khomh, and Antoniol (2022), as it has been found to scale better for complex CPSs Nejati *et al.* (2019). Other approaches tackle the test oracle problem in the context of CPSs Menghi *et al.* (2019); Ayerdi *et al.* (2021, 2020a, 2022). There is another large group of studies that target regression testing of CPSs with the goal of cost-effectively testing such systems by selecting appropriate test cases and prioritizing them Birchler *et al.* (2022); Shin *et al.* (2018); Arrieta *et al.* (2019b, 2023, 2019a). Our approach is complementary to all of these, i.e., any of these approaches can be combined with our framework. The core difference is that this paper, instead of focusing on the generation of test artifacts (e.g., test cases, test oracles) and on their selection and/or prioritization, focuses on their execution by leveraging a microservice-based architecture that allows for (1) executing test cases and test oracles across all test levels and (2) integrating the approach within a DevOps ecosystem for CPSs Gartziandia (2021).

For the case of CPSs, different approaches propose test architectures for CPSs modeled in MATLAB/Simulink (Arrieta *et al.*, 2017a; Zander-Nowicka, Schieferdecker, and Perez, 2006). However, in the field of CPSs, microservices for testing are still in their infancy. Gartziandia (2021) proposes a microservice method based on Machine Learning algorithms to discover execution issues in CPSs in run-time when new software versions are deployed. We have not used this kind of algorithm. They demonstrate that their approach reduces time and effort of deployment by automating the process, and reduces maintenance costs like in our approach. Moreover, they affirm that an architecture based on microservices offers the flexibility to adapt to different test levels and installations, and permits scaling the system. Our approach differs from Gartziandia (2021) in different perspectives. On the one hand, our microservice-based framework is focused on design-time testing instead of run-time testing. On the other hand, our microservice-based framework is focused on functional properties of the system, whereas Gartziandia (2021) focuses on performance faults.

The problem of test re-usability across different test levels (e.g., MiL, SiL and HiL) is not new. This is well-acknowledged by industry, for which different interoperability standards have appeared. For instance, in the context of CPSs, the ASAM-XiL (ASAMRef.) standard is a well-known standard, which has been acquired by key CPS simulation vendors (including, but not limited to Mathworks, National Instruments and dSPACE). This permits re-using test cases at different test levels at design-time. However, it does not permit the re-usability of test cases and test oracles at operation, something that our approach permits.

4. Case study

4.1. The system

Orona is a company dedicated to the elevator industry. Orona designs, manufactures, installs, and maintains elevators, escalators and moving ramps. A system

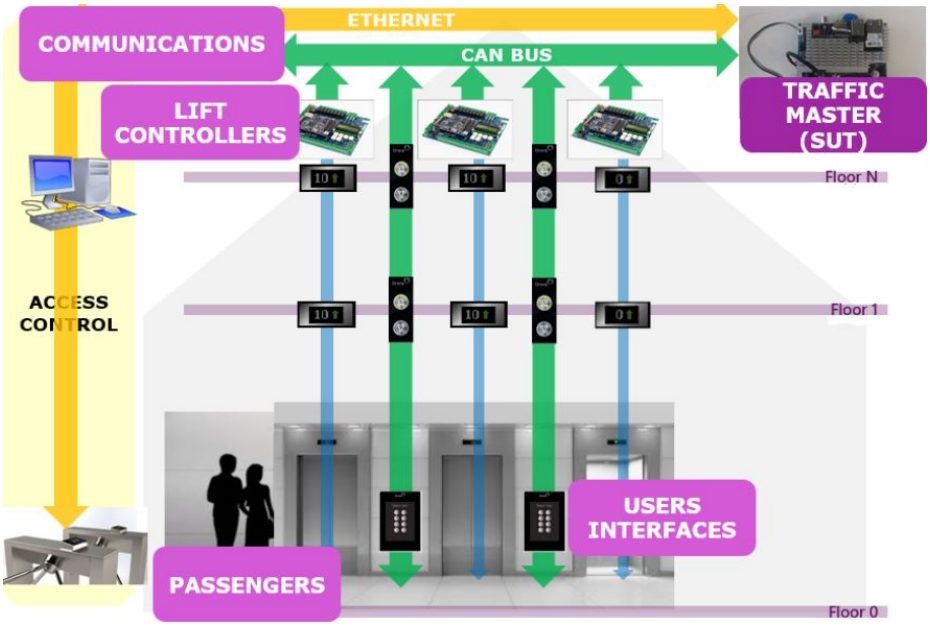

Figure 1. Overview of the architecture of the Elevators installation

of elevators aim at transporting passengers from a floor to another safely while trying to guarantee the maximum comfort as possible. A passenger arrives at a floor and calls for an elevator through a push button. This button can be either conventional (i.e., provides an up or down direction) or with destination control (i.e., provides the specific floor at which the passenger aims to travel). This information is sent to the traffic master, which is the case study of this paper. The traffic master receives this floor call, and, by considering different information (e.g., where each elevator is, which is the weight of each elevator), it provides an assignation to the call. In advanced buildings, this traffic master also encompasses information about access control, i.e., where each passenger can travel to. Figure 1 presents an overview of the elements of this CPS.

4.2. Development and Operation Methodology

When the simulation of the traffic study is accomplished, the selection and configuration of the different scenarios is complex and time extensive. In addition, there is not any automatic feedback mechanisms to detect problems. As a consequence, when problems arise, the time and effort is significant. Validation and deployment of the system are also semi-manual. As for validation, information from the operation is not accessible, so it is not possible to reproduce real situations in the laboratory, and the decision of whether a test case has succeeded or not is manual. With regard to the deployment, the maintainer is responsible for configuring and updating new versions in the installation, which is also a manual process. In this paper, we present the mechanism to reduce the time needed and consequently, the effort of the simulation process. Furthermore, we

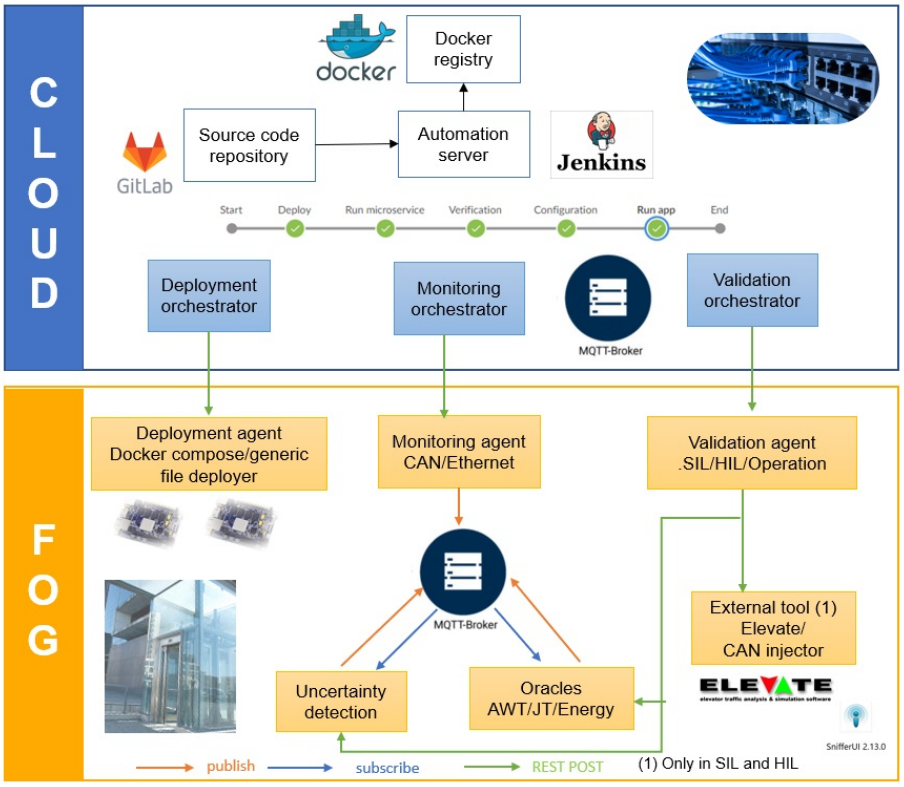

Figure 2. Overview of the architecture for DevOps of the Traffic Master (Arrieta *et al.*, 2022)

will analyze the economic savings with the implementation of the new process (consequences of the problem analysis in the section 2).

The traffic master is constantly evolving for different reasons (e.g., add new functionalities, deal with hardware obsolescence, bug corrections). As a result, Orona, through the Adeptness (H2020) project, is aiming to establish a DevOps pipeline that aims to automate the continuous delivery of new software versions (i.e., when a change in the code is completed and committed, the automated test at different levels, automated deployment, run-time monitoring and recovery in case the new version fails in operation). Because of this, our multi-level testing framework had to be integrated with their DevOps methodology, which is based on microservices. Figure 2 presents an overview of the proposed architecture. The architecture includes microservices at two computational levels: (1) the cloud and (2) the fog, i.e., a local network close to the lift. The architecture can be divided into four main subsystems: (1) the automation server, (2) the deployment subsystem, (3) the monitoring subsystem and (4) the validation subsystem (Arrieta *et al.*, 2022). This paper targets the validation subsystem of the DevOps ecosystem. Sections 6, 7 and 8 describe the validation orchestrator, validation agent and the external tool of the architecture.

5. Verification and Validation Test Benches for CPS

OMG's (Object Management Group, Inc.) working group developed in 2001 a UML (Unified Modeling Language) configuration file designed for model-based testing, called UML Test Configuration File 2 (UTP2). UTP2 is utilized to design, visualize, specify, analyze, construct and record common and required artifacts in various test methods (especially model-based testing) (UMLTesting-Profile, 2018). As UTP2 is part of UML, UTP2 files can be used to communicate artifacts from different ecosystems, facilitating the labour of different classes of engineers.

UTP2 has been designed for users with model-based testing knowledge. These experts can be system/software engineers, stakeholder, certification authorities test engineers, requirements engineers and domain experts. They will be in charge of designing test models of different test levels, multi-level testing to address non-functional requirements, integrate with SysML to achieve requirement traceability and test generation, comply with relevant industry standards and the ability to reuse test logs for further test evaluation (UMLTestingProfile, 2018).

UTP2 plans control the existing components and requirements, which can be used to record, evaluate test ranges and calculate progress reports. At the end, UTP2 tries to extract as much information as possible from test plans. Test cases are used to evaluate the test quality of the project and verify whether the test project meets its specifications.

In this project, UTP2 standard has been used to define the concepts to define the multi-level testing plan for CPSs. The elements needed in our validation plan are the following:

- ValidationPlan: The validation is usually the document that contains the planning for the validation system. The main element is ValidationPlan for this reason.
- ValidationPlan_Id: each ValidationPlan contains a unique identifier.
- TestCases: The UTP2 document defines a Test Case as a procedure that includes a set of preconditions, inputs and expected results, developed to drive the examination of a test item with respect to some test objectives. In the JSON, each test case contains this information. The validation might contain more than one Test Case.
- TestCase_Id: a unique identifier identifies every single test case.
- TestCase_Priority: each test case has a different priority, and this must be included in the JSON.
- TestCase_TestLevel: there are different levels in each test case (HiL, SiL in our case)
- SUT: It provides information about the version of the system that is being tested.
- Version_Id: a unique identifier identifies every single SUT.
- Configuration_Id: every single SUT contains a document that provides information about the initial configuration.
- TestArtifacts: is defined as an object produced or modified during the execution of a process. It is composed of different test inputs.

- TestDesingInput_Id: each Test Input contains a unique identifier that provides information about the name of the document that will be used with the input values.
- TestDesingInput_Type: each test input file could be in a file with different formats. This value provides information about the format of the document.
- Oracles: is a mechanism, different from the program itself, that can be used to verify the correct functionality of a system. We may have more than one oracle.
- Oracle_Id: a unique identifier identifies every single oracle.
- Oracle_Type: the multi-level testing framework allows for three types of oracles: Assertion, Metamorphic and Machine Learning-based test oracles. However, only Assertion and Machine Learning-based oracles are applicable to run-time.
- Oracle_Criticity: the aim of this field is to inform the importance of the test failure. If the test is important, the criticity will be high. As a consequence, if one test fails, we will consider that the test has not passed.
- Oracle_Inputs: it contains the information about the input values needed to execute the oracle.
- Oracle_Verdict: it informs about the final decision about the test, if it fails or passes.

The validation orchestrator microservice is in charge of managing the whole lifecycle of a validation plan in the context of the DevOps ecosystem. Developed in Python, the validation orchestrator is capable of running in the major platforms where the Python interpreter or Docker is supported. All the communications are made exclusively through APIs. The validation agent microservice is responsible for coordinating the execution of a validation plan and components in an evaluation platform (HiL, SiL, an installation in operation, etc.). As it happens with the validation orchestrator, the validation agent is also developed in Python, sharing the ability to be run on the majority of platforms.

6. Validation Orchestrator

In this section, we explain a generic Agile Testing Framework. The framework implements microservices of one specific template that allows for integrating our framework with other microservices that together tackle DevOps for CPSs (Gartziandia, 2021). Figure 3 shows the general DevOps architecture for CPS, and it focuses on (1) the validation orchestrator; (2) validation agent and (3) external tool microservice (a convenience microservice developed with the integration of legacy, unsupported or out of scope tools into the project ecosystem).

6.1. Architecture

The validation orchestrator microservice is in charge of managing the whole lifecycle of a validation plan in the context of our ecosystem. The validation

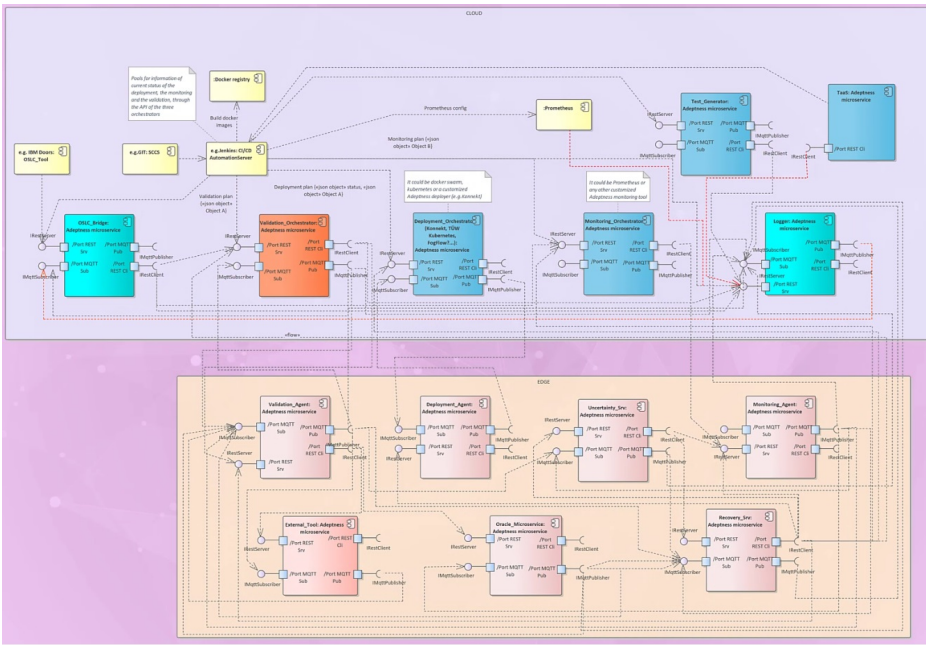

Figure 3. All the elements of the architecture

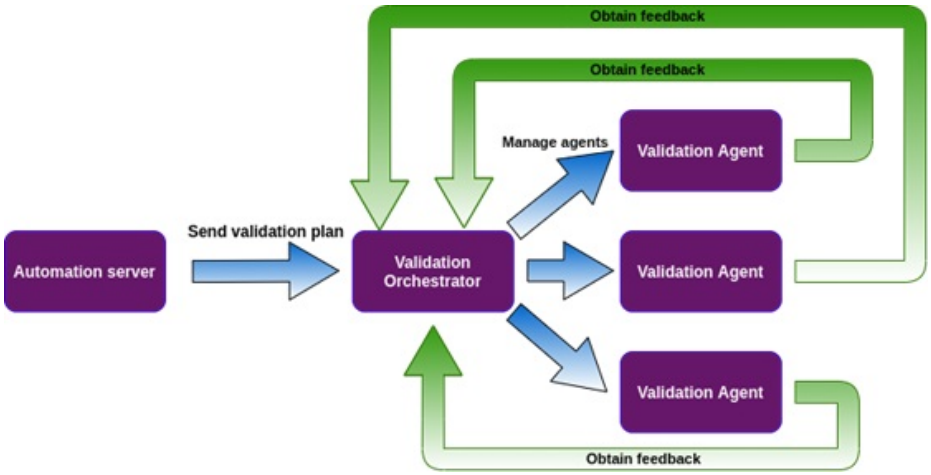

Figure 4. Validation orchestrator schema

orchestrator is capable of running in the major platforms where the Python interpreter or Docker is supported. On request by the automation server, the validation orchestrator receives a validation plan containing all the required information for performing a validation procedure via a REST API call. The validation orchestrator interprets the validation plan, requesting additional information of the components involved in the validation if necessary, and generates

sub plans that contain the validation procedures to be delegated to each of the validation agents (see figure 4).

The validation orchestrator is divided into four main blocks: The microservice core, the parser, the agent manager, and the aggregation rules package.

The microservice core, developed on top of a microservice template (Gartziandia, 2021), is responsible for managing the interaction of the orchestrator with the other components in the DevOps ecosystem. The microservice core implements the REST API endpoints exposed by the orchestrator, and serves as the entry point for accessing the functionality provided by the orchestrator. A microservice template in our context encompasses a set of basic asynchronous and synchronous communication endpoints. These are provided regardless to the role of the microservice. These endpoints offer basic information about the execution status, performance and health. On the one hand, the synchronous interfaces allow other services to request microservices' health status. On the other hand, asynchronous interfaces allow microservices to publish relevant data without knowing the receiver of the messages.

When a call is received on the validation plan setting endpoint, the microservice core passes it to the parser component of the microservice. The parser component is responsible for creating an internal representation of the elements represented in the validation plan. As the structure and elements of the validation plan are shared among the validation agent and orchestrator, the parser and the Python class hierarchy have been extracted to an external library, that is shared and used between the validation agent, orchestrator, or any other component that needs to make use of components represented in the validation plan. After parsing the plan, the parser lends the parsed plan to the validation agent manager. The validation agent manager is responsible for managing the validation agents that are required for the execution of the validation plan. Each of the validation agents is responsible for managing a test level on the edge.

From the main validation plan, the validation agent manager will extract the subset of elements that need to be managed by each of the validation agents, and create sub-plans to be sent to each of the validation agents. The validation agent manager will then perform the required REST API calls to configure and start the validation on the validation agents, and subscribe to the topics where the verdicts generated by each of the validation agents will be published.

When receiving a verdict, the validation orchestrator determines the source agent for the verdict, stores the received verdict and performs an aggregation procedure for the verdicts. The aggregation procedures allow grouping verdicts raised by lower level components, and generate verdicts for higher level validation elements. In the DevOps ecosystem, the evaluation starts from oracles, which generate verdicts on their own, and then are aggregated to generate a verdict common to the test case, continuing with the validation level, and finally a global verdict for the validation plan. Several strategies have been developed for aggregation:

- Default Aggregation: The default aggregation strategy consists in marking the global verdict as passed when all the aggregated verdicts have passed. Any failed verdict would make the aggregated verdict to fail.

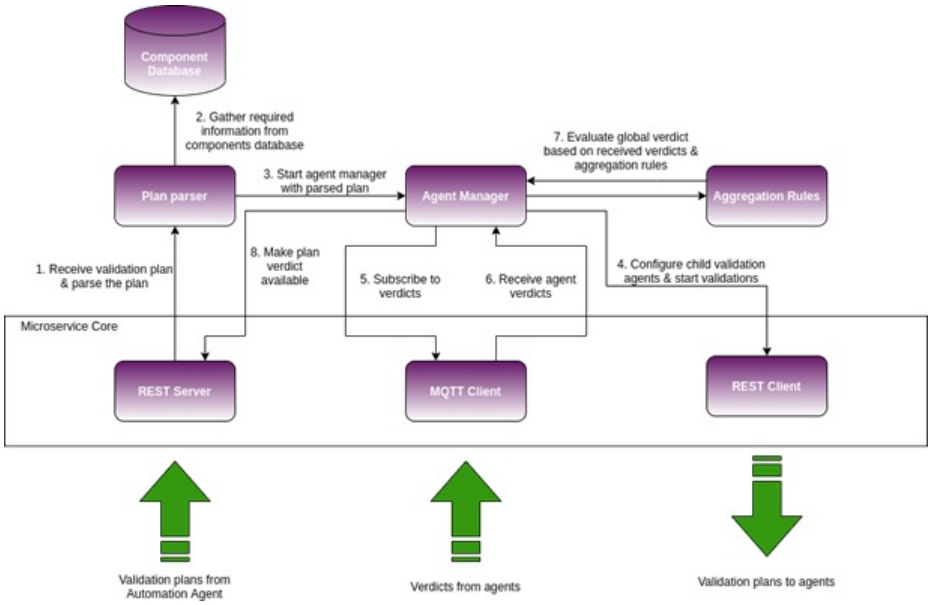

Figure 5. Validation orchestrator validation plan execution workflow

- Percentage Aggregation: Marks the aggregated verdict as passed if a certain percentage of the aggregated verdicts have passed.
- Criticity Aggregation: Marks the aggregated verdict as passed if all the critical oracles have passed.

6.2. Deployment

The validation orchestrator is shipped as a Docker container that is deployed on the cloud. A new Docker image is automatically generated each time a new commit is uploaded to the code repository via Gitlab CI/CD pipelines. As part of this process, the validation orchestrator passes several QA procedures, by performing unit tests and automatically generating the documentation for the project with the Sphinx documentation generation tool.

A single instance of the validation orchestrator manages all the validation agents that are running on the edge. The validation orchestrator shares the same requirements as any base microservice from our related DevOps ecosystem, being able to be executed on any platform supporting Docker.

6.3. Test execution workflow

For the execution of all validation plans, the validation orchestrator follows a well-defined execution workflow that includes the interpretation of the validation plan, the configuration of child components and recollection of verdicts raised by the validation agents. Figure 5 shows a quick insight on this execution process.

6.3.1. Parsing of the input validation plan

The validation orchestrator exposes a REST endpoint for receiving validation plans. This endpoint receives a JSON formatted validation plan, and performs a parsing procedure for interpreting the contained validation elements.

The parser performs an initial interpretation of the validation plan and its contents. In the case of the validation orchestrator, this parsing involves determining which are the validation agents to be configured, and which is the aggregation procedure to be followed for combining the received verdicts and raising a global verdict for the validation plan. Given that the validation procedure is delegated to the validation agents, the validation orchestrator does not interpret the lower-level validation elements, leaving that interpretation to the parser on the validation agent. Thanks to this abstraction, the validation orchestrator is agnostic to changes or additions in the lower layers of the validation plan.

6.3.2. Obtaining additional configuration from external data sources

In some cases, it is possible that during the interpretation of the validation plan, additional information about the validation agents could be required. This information could include dynamically configured endpoints required to configure or communicate with the validation agents. The validation orchestrator has been developed with such cases in mind, and provides extendable abstract implementations in the validation plan parser for communication with additional data sources.

6.3.3. Agent configuration and plan execution

When all the information from the validation plan has been parsed and understood, the validation orchestrator determines the validation agents that will be used to run the validation, and starts the process of configuring the agents for the validation procedure to be managed by each of them.

First, the subset of the validation plan corresponding to the agent is rendered as a JSON. Once the validation plan to be sent to the validation agent has been generated, the validation plan is sent to the agent. This process is repeated for each of the validation agents, sending the corresponding validation plan subset for each of the agents.

6.3.4. Verdict arrival, aggregation and publication

The validation orchestrator awaits verdicts published by validation agents through MQTT. Upon reception of a verdict, the validation orchestrator checks the aggregation rules defined on the validation plan, and computes a global verdict according to the aggregation rules. If a global verdict is obtained from the aggregation, the global verdict is published.

We selected MQTT because it fitted our needs (e.g., soft-real-time communication with monitoring variables, etc.). With MQTT, it is possible to asynchronously provide data to other related microservices via a publish and subscribe method. For instance, the verdicts of the test oracle are consumed by other tools in order to be monitored.

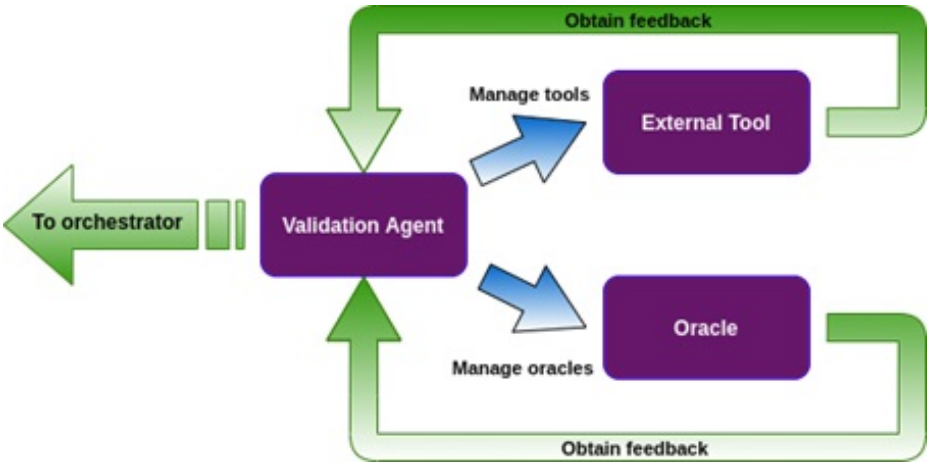

Figure 6. Validation agent microservice interaction overview

7. Validation Agent

In this section, we continue with the explanation about the generic Agile Testing Framework developed for the DevOps ecosystem (Gartziandia, 2021). The framework implements microservices of such an ecosystem for the validation agent. The validation agent will coordinate the Oracles, and external tools required to perform an evaluation, configuring them, managing the execution and collecting the generated test results. The generated results will then be aggregated and published, so that the validation orchestrator is able to collect the results (see figure 6).

7.1. Architecture

The validation agent is divided into five main blocks: (1) the microservice core, (2) the parser, (3) the oracle manager, (4) the external tool manager and (5) the aggregation rules package.

The microservice core is developed by making use of the APIs exposed by the microservice template, building the REST API endpoints. Calls for managing the execution of the validation plan and the validation agent and retrieval of plan verdicts have been developed. The callbacks for each of the REST API endpoints interact with the parser and managers to implement the evaluation and execution logic.

The parser is responsible for interpreting the validation plans received from the REST API endpoint. The parser is on charge of transforming the input payload to classes that maintain all the attributes specified in the validation plan. It is important to mention that it contains the logic to transform the structures to be transferred to the child components. In the validation orchestrator and the validation agent, all the structures are defined in an external library shared between the validation orchestrator and validation agent in order to maintain

consistency between the structures. The validation orchestrator and agent are less prone to integration issues thanks to the definition of a shared library.

The parser expects a JSON structure containing the URN (an identifier that will uniquely identify the validation plan) for the parent validation plan, aggregation section and a tests section containing each of the test cases to be run. For each of the test cases, a SUT section, an oracles section, external tools section, and aggregation sections will exist.

The oracle manager is responsible for managing the child oracles, their execution and the recollection of verdicts raised during the evaluation. Upon plan reception, the oracle manager extracts the oracle structures from the validation plan. Each of the oracle structures contain a section defining the MQTT and REST API endpoints of the oracle. Additionally, it has an inputs structure that contains the binding of each of the supported inputs on the oracle to monitors and sensors and an additional oracle settings structure containing additional settings for the oracle. For each of the supported inputs, the oracle structure contains the monitor and sensor where the data will be published.

The oracle manager subscribes to the verdict topic of each of the oracles and awaits verdicts raised by the oracles. When a verdict is received, the verdict is stored on the internal structure for each of the oracles and the aggregator is notified for performing the recalculation of the global verdict for the agent.

The external tool manager is responsible for managing the external tools defined on the validation plan. For each of the external tools, calls to set up the tool and manage its execution are issued. The internal structure of each external tool contains an endpoints section that contains endpoints for the external tool to be configured, the tool type, and an additional tool settings section containing specific configuration for each of the external tools.

Finally, the aggregation rules module is responsible for aggregating the received oracle verdicts and generating a global verdict for the validation plan being run in the validation agent. The aggregation is made according to the rules defined in the validation plan, with the same aggregation strategies as in the validation orchestrator available.

7.2. Deployment

The validation agent is also shipped as a Docker container. A new Docker image is automatically generated each time a new commit is uploaded to the code repository via Gitlab CI/CD pipelines. As part of this process, as it happens with the validation orchestrator, the validation agent also passes several QA procedures, by performing unit tests and automatically generating the documentation for the project with Sphinx.

The validation agent is deployed on the edge, having one validation agent per test level (HiL and SiL). The validation agent manages all the validation components of that test level, with the ability to manage components that are deployed in the same node or in other nodes that also conform the same test level, as long as the agent is able to reach the endpoints of the managed components.

The reason for separating the validation process between agent and orchestrator is twofold. First, thanks to this responsibility separation, the validation

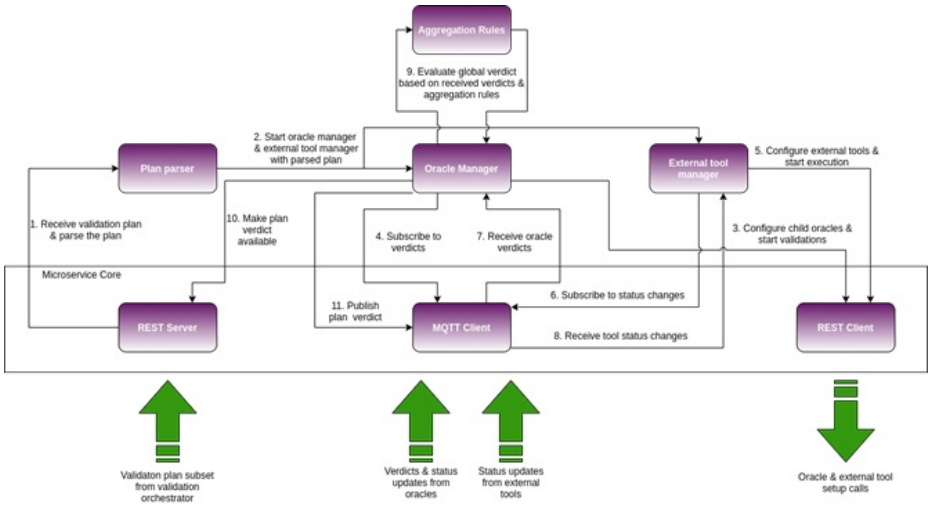

Figure 7. Complete execution flow for validation agent

orchestrator is agnostic to the validation components being run in the agent, simplifying the orchestrator overall, and focusing the orchestrator in the coordination of high-level components. Secondly, having the agent on the edge allows for improved communication latencies between the managed components and the agent, reducing the issues that could arise from incorrect timing in validation components, reduce traffic to the cloud, having the need to only forward global verdicts to the orchestrator. Finally, reducing the need for the exposure of REST endpoints to components outside the edge, improving the security posture of the test level.

7.3. Execution flow

In the following section, the execution flow followed by the validation agent for carrying out a validation plan and coordinating child oracles and external tools will be described. This execution starts with the reception of a validation plan from the orchestrator, continues with the configuration of child validation components and finishes with the recollection of verdicts, aggregation and forwarding to the validation orchestrator. On the figure 7, an overview of the mentioned execution flow is shown.

7.3.1. Reception of validation plan

Initially, the validation agent is in idle state, waiting for reception of a validation plan from the orchestrator to be executed. When the orchestrator sends a validation plan to the validation agent endpoint, the validation agent starts the interpretation and execution of the received plan.

Initially, the validation agent will forward the validation agent to the parser component, and the parser will interpret the plan and store it as an internal representation. This representation will include the test to be run, with the

structures for each of the child oracles and external tools, their configuration, and verdict aggregation strategies.

If the parsing process is successful, the validation agent will send a positive response to the orchestrator and continue with the next phases in the execution of the validation plan.

7.3.2. Child component configuration

After the interpretation of the validation plan, the agent and oracle managers will be notified, and the configuration of each of the child components will start.

The oracle manager will extract the oracle structures from the tests in the parsed validation plan, and for each of them, the oracle manager will send calls to set up the oracle.

The external tool manager follows a similar approach as the oracle manager does for configuring oracles. When a new validation plan is received, the external tool manager will be notified, and the configuration process for the required external tools will begin. The external tool manager will extract the required external tools from the validation plan, and send configuration requests.

7.3.3. Validation plan execution

After all the child elements have been configured, the validation procedure will start. For this, both the oracle manager and the external tool manager will request status changes for oracles and external tools. Each of the components will answer with a positive response if the status change has been successful. The oracles will be the first components to be started, as the external tools might cause the publication of values to be evaluated. This is the case with the Elevate external tool, as the executed simulations trigger monitors that publish operational values to be evaluated by oracles.

After all the oracles have been started, the external tool manager will be requested to send status change requests to the external tools.

7.3.4. Verdict arrivals & aggregation

As part of the configuration procedure, the oracle manager will subscribe to verdicts to be raised by child oracles. When an oracle verdict is received, the origin of the verdict will be determined, and stored in the respective oracle internal structure. After the verdict is stored, the corresponding verdict aggregators will be called. First, the aggregator for each of the test structures will be called with all the verdicts for the oracles in that test case. If a verdict for the test case is obtained, the test case structure will be updated with the updated verdict, and the aggregator for the validation plan being run in the agent will be called.

7.3.5. Submission of verdicts

If the top-level aggregator reaches a verdict for all the test cases, the global verdict will be stored in the validation plan internal structure, and then forwarded to the validation orchestrator, which will run the corresponding aggregation for the parent validation plan.

8. Instantiation of the agile test execution framework on the elevation case study

With the goal of demonstrating the feasibility of the approach with our industrial use-cases, this section explains the instantiation of the DevOps ecosystem within the SiL test level of Orona's use-case (i.e., the elevation use-case).

8.1. Introduction

The external tool microservice is a convenient microservice developed with the integration of legacy, unsupported or out of scope tools into the DevOps ecosystem (Gartziandia, 2021). The external tool microservice exposes a basic API and means of managing the execution of a sub-process or external application. This API can be extended to provide additional functionality and means to manage the tool being commanded by the external tool microservice. For each tool that wants to be integrated on the DevOps ecosystem, an ad-hoc implementation of the external tool microservice is performed, creating external tool subtypes for each of the integrated tools.

In the context of CPSs, testing is usually carried out through simulation-based testing Arrieta *et al.* (2019a,b, 2023); Briand *et al.* (2016); Birchler *et al.* (2022); Menghi *et al.* (2019, 2020). Therefore, the external tool microservice would provide communication with one such simulator. In the context of our industrial case study, such a simulator has been Elevate, a domain-specific tool for elevator dispatching algorithms. However, in other CPSs, this could be adapted to support other simulators, e.g., for general purpose software systems MATLAB/Simulink Arrieta *et al.* (2019a,b, 2023); Menghi *et al.* (2019, 2020), for autonomous driving systems BeamNG Birchler *et al.* (2022).

8.2. Architecture

To instantiate the Agile Test Execution framework within one of the industrial case study, we have developed two external tools to execute test cases at the SiL and HiL test levels respectively. The base external tool microservice is built on top of a Python microservice template (Gartziandia, 2021), and it exposes basic API endpoints for managing the execution of a command launched by the Python sub-process library, and receiving feedback about the execution status. Figure 8 shows a high-level overview of the external tool for the SiL test level, which is the one developed ad-hoc for our case study.

Figure 9 shows the sequence diagram of the process. The validation agent will configure the external tool microservice, defining the process to be launched. It will then start the execution of the desired process and expose its interaction paths. The validation agent will interact with the launched process through these paths, collecting data from the external process or sending data for interaction.

8.2.1. REST and MQTT API endpoints

The external tool microservice provides a very basic API to manage the execution of the tool being managed by the microservice. The management of the

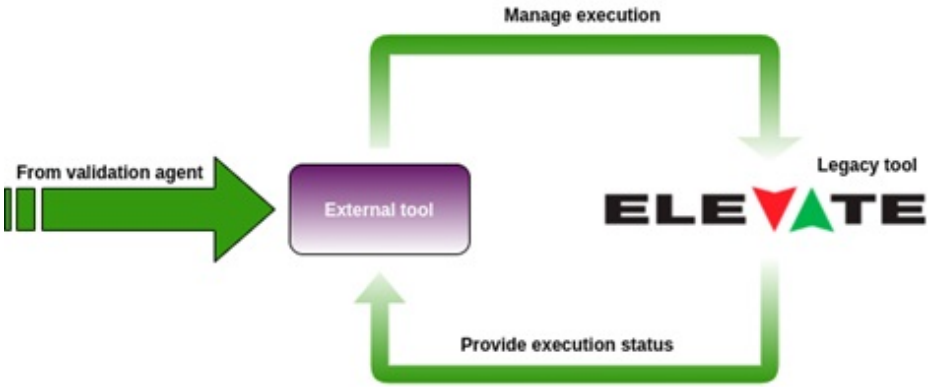

Figure 8. Interaction of the external tool microservice with other validation components

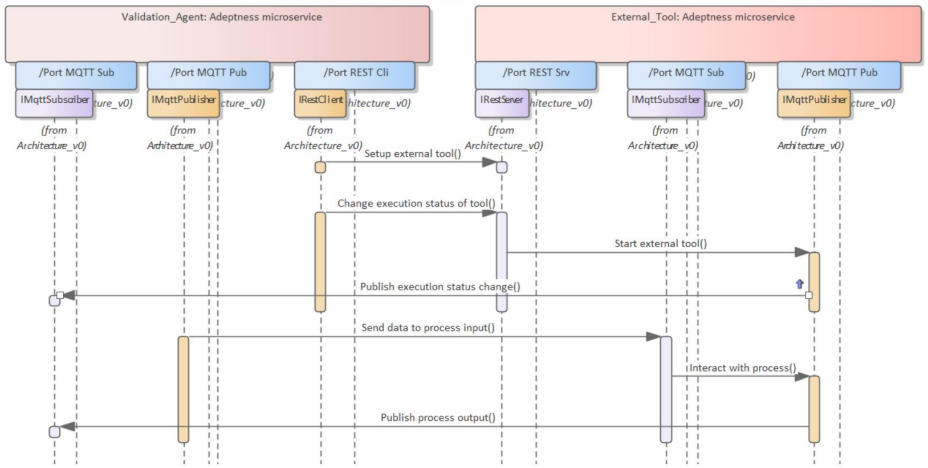

Figure 9. Sequence diagram of the external tool execution

execution status is performed determining the status, which can be "running" or "stopped". The execution status can be consulted following the same process. This would be the same for any kind of simulator, regardless the type of CPS being used. Therefore, this decision is not ad-hoc for our case study.

8.2.2. Internal architecture

The external tool is divided into the microservice core and a sub-process manager component, that is in charge of managing the execution of the sub-process launched by the external tool. When receiving a status change request, the request is forwarded to the sub-process manager, which launches a predefined shell command in a separate thread, and waits for completion of the execution. Depending on the tool to be implemented, additional care has to be taken, as some legacy applications tend to run in background and return the control to the parent process, causing the standard sub-process library to mark the process

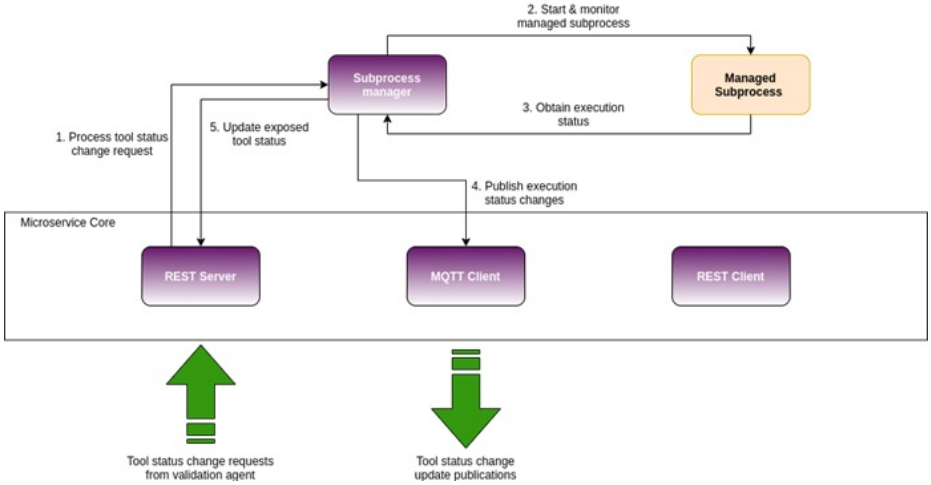

Figure 10. Execution flow for the validation agent on the elevation case study

as terminated. This was the case of our case study, and could also be the case of other CPS simulators.

8.3. Execution flow

The external tool execution flow starts with the reception of a status change request from the validation agent, and continues with the execution of the contained tool and the reporting of the status changes. The following diagram (figure 10) shows an insight of this execution process, which would be applicable to any kind of CPS simulator.

The external tool microservice awaits for status change requests. When receiving a status change, the external tool starts its execution flow.

When the status change request is received, the external tool microservice checks the current status of the sub-process. If the sub-process is idle, the external tool will proceed and request the sub-process manager to change the tool status.

8.3.1. Subtool execution

Upon request, the sub-process manager will start a new thread to manage the sub-process, and launch the predefined command. The execution of the called command is synchronous, so the thread launching the command will wait until the termination of the tool. In the cases that the launched command immediately returns control to the launching application and runs in the background, a different approach has to be followed. In these cases, the sub-process manager provides the ability to launch the process and monitor the process.

8.3.2. Status reporting

The thread launching the tool will be in charge of updating the tool execution status. Upon launch, the thread launching the external command will update the internal structure storing the tool execution status, and perform a publication.

8.4. Application to the elevation use-case

In the Elevation use-case, Orona makes use of a legacy tool named Elevate. Elevate is an elevator installation simulator, used by Orona as a platform for validating developed traffic algorithms at SiL level. Elevate receives a configuration file that defines the installation characteristics (e.g., building level quantity, number of elevators, dynamic information of the elevators (e.g., speed, acceleration, jerk)) and a passenger list file, containing a list of "passengers", represented as calls to the elevators at different building levels and times. The passenger list can be built based on traffic captured from a real installation or based on simulated statistical models.

In our industrial case study, the SUT is the traffic algorithm being developed for Orona's elevators. This algorithm is tested at the SiL test level using Elevate, which supports loading an external traffic algorithm. As Elevate is a key part of the validation workflow at Orona, it was desired to be able to automate the execution of test cases with Elevate on the DevOps ecosystem (Gartziandia *et al.*, 2021), so an external tool wrapper for the Elevate simulator has been developed.

In the case of HiL, Orona uses a proprietary software that communicates with the HiL test bench. The external tool was also adapted to manage the execution at the HiL test level. As this tool is proprietary, due to Intellectual Property issues, we avoid giving further details in the paper.

8.4.1. Elevate external tool

The Elevate external tool is an ad-hoc implementation of the external tool microservice, adapted to the Elevate simulator application. Based on the reference implementation for the external tool, it provides an improved wrapper over the Elevate simulator.

Elevate is a Windows GUI application running on 32 bits. On launch, the application creates a background thread and returns control back to the terminal, meaning that additional checks need to be made to check the execution status of the tool. Elevate is a perfect candidate for being wrapped in a microservice layer that ideally will serve as a Simulation-as-a-Service tool for Orona.

8.4.2. Execution of the tool in validation example

Figure 11 shows an example execution of the Elevate application within our integrated environment. A physical Windows machine where Elevate is installed is used for launching the external tool microservice, where it awaits connections. After the external tool microservice has received a request from the validation agent, we can see how the microservice answers the validation agent with a

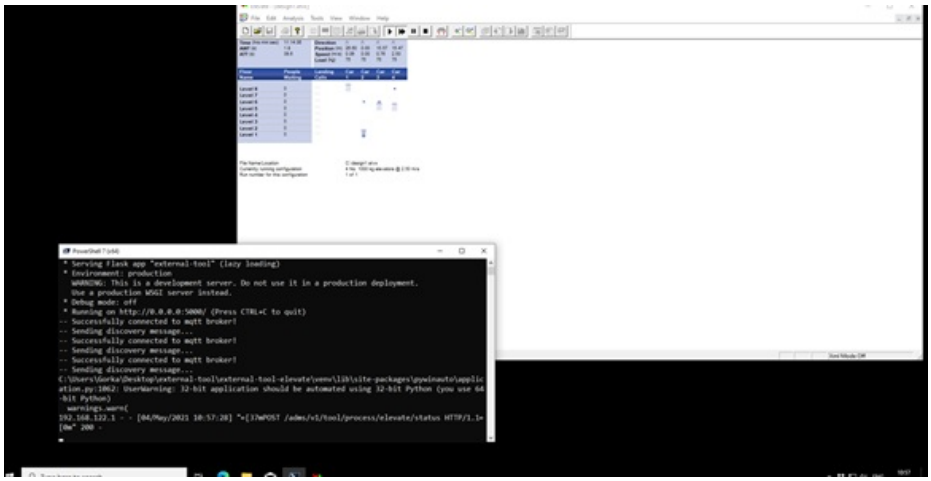

Figure 11. Execution of the elevate simulator with the external tool

positive response, after which the Elevate simulator is launched. In this case, the building configuration file and passenger list file are already placed on the target launch directory, and the target traffic algorithm SUT placed on the Elevate installation folder as a DLL file. While the application is running, operational values will be posted and collected by the oracles, that will perform the evaluation and raise a verdict for the tested conditions.

9. Evaluation

This section evaluates and compares the previous approach in which tests were generated, executed and checked manually with the actual automatic approach. Our solution impacts on the consequences (economic and time cost) of the problem analysis described in the section 2. Wang *et al.* (2015) demonstrate due to different experiments that CPS are time-consuming, needing a significant manual exertion before they can be utilized as HiL experiments. Thanks to our contribution, we will demonstrate that we can reduce the time needed. Furthermore, we will demonstrate the reduction of economic cost following the models proposed by Wang *et al.* (2015) to calculate the cost of CPS. Research questions are:

- RQ1: Has the time needed to execute and evaluate the tests been reduced?
- RQ2: Has the economic cost needed to execute and evaluate the test been reduced?

In this section will be described all steps needed in both approaches (manual and automatic). Table 1 shows the differences in each step between the previous manual approach and the actual automatic approach.

Table 1. Comparison of the automatic process with the previous manual multi-level testing process

Steps	Manual	Validation workflow	Difference
Step 1	Manual test configuration and manual input creation	Manual test configuration and manual input creation	There are no differences
Step 2	Manual test definition	Automatic test definition	In this step, tests that will be executed are created. In both cases, the configuration of the tests that will be executed and their input values are selected besides, the algorithm that will execute the tests. The difference is that previously this was executed manually and now automatically. Moreover, the automatic test definition includes the oracle configuration and the validation strategy
Step 3	Manual Elevate execution	Automatic Elevate execution	The automatic mode executes Elevate automatically instead of manually
Step 4	Manual result evaluation	Automatic result evaluation	Previously, all results were analyzed one by one by a person comparing the obtained results with the expected results saved in a table. The automatic process checks these results automatically thanks to oracles

9.1. Comparison with manual test system generation

The steps needed to generate the test system are similar in the manual and in the validation workflow or automatic approach. Nonetheless, there are some differences between both approaches, which have been briefly described in table 1. This section will explain those differences with more detail.

9.1.1. Step 1: test configuration and input creation

The first step corresponds to the creation of the test configuration and inputs. This step is exactly the same in both approaches. This process is completely manual and time-consuming, but necessary. The first action is the creation of a file containing all the possible tests that can be accomplished, the configuration of the tests. This procedure is important due to the fact that all possibilities might be included to be used in the execution step when they are necessary. After the creation of the configuration test, a file including all possible input values is created. This procedure is also important to check all possibilities, and it is also manually carried out in both approaches.

9.1.2. Step 2: test definition

The second step corresponds to the definition of the tests that will be executed. The manual approach includes three actions. The first action is the definition and selection of the tests that will be executed from the configuration file created in the step before. A test group is selected from the group created in step 1. The selection process is based on a state-of-the-art technique.

Afterward, the second action consists of the selection of the appropriate inputs for those selected test from the configuration file from the step 1. The suitability of the test inputs is decided based on the test scenario and configuration to be tested (installation and number of elevators) and on the traffic profile that better exposes the functionality to be tested (Ayerdi *et al.*, 2020b). The final and third action consists of the selection of the algorithm that will be used in the step 3. This algorithm is contained in a Windows DLL library built with an SDK provided by Elevate, being the SUT to be tested. While the selection of the test inputs is relatively trivial based on the selected test configuration, the preparation of the test input can span over a complete day. Usually, the traffic profile is collected from a real installation, by recording the hall and cabin calls and extrapolating the passengers count and direction based on other measures, such as from the lift weight meter (Siikonen, 1993).

The validation workflow implements the same action differently, and it includes additional ones. In the automated validation workflow, the test definition and the appropriate input selection continues to be performed manually. However, thanks to the standardized manner of defining the test strategies as validation plans, the validators are able to reuse the defined test strategies in different test scenarios or in different executions of the same validation. Validators define the validation JSON manually or assisted by tools that allow selecting the test inputs and oracles to be used from the available ones. Once the validation plan is defined, this plan can be stored and retrieved for posterior uses.

9.1.3. Step 3: Elevate execution

On a manual workflow, the execution of Elevate is made manually as a normal Windows GUI application. On the GUI, several steps for configuring the simulation have to be performed. From different menus and dialogs, the algorithm to be tested, the configuration of the simulated building, and the selection of the passenger profile to be used as test input has to be made. After the configuration phase, this configuration can be saved in an .elvx file that can be later opened to load the settings on Elevate. After the configuration phase, the execution of the simulation is performed. This simulation can be executed in real-time, or in accelerated time, where the execution step is limited by the performance of the computer on rendering the simulation and GUI elements. Finally, the results are shown on the GUI, leaving the user with the option of exporting these results into PDF or CSV files. Additionally, if logs have been defined on the algorithm, a log file is also stored on the execution path of Elevate, having to also manually collect and interpret this log file.

On an automated workflow, the execution of elevate is made automatically from a command line interface, thanks to a built-in option to run simulations on batch. While still being experimental, this option instructs Elevate to scan the root path of the current hard disk drive for already defined .elvx files. For each .elvx file, Elevate runs the GUI and executes the simulation in accelerated speed, storing the results of the simulation and logs on the root of the hard drive.

9.1.4. Step 4: Checking multi-level testing results

In the last step, the test evaluation results are checked. In the manual mode, the results obtained in the evaluation are checked with the results included in a document. This procedure is carried out by a person checking each result one by one. The validation workflow performs this procedure automatically, checking each result by an algorithm instead of a person. This reduces the time needed to check the evaluation drastically. Manual tasks include the comparison of QoS metrics with previous simulations, manually reviewing each results file, or interpretation of logs to verify values of internal variables of the algorithm, performing the interpretation and evaluation of these values manually, based on previous knowledge on the algorithm and code. On the automated workflow, oracles automatically perform the required evaluations and publish verdicts that are automatically aggregated until a global verdict is achieved.

9.1.5. Overall test system generation time

Based on the above-mentioned steps, we separated the time required by each approach in two algorithms; one for the manual approach and another for the automatic validation workflow. The time required to manually execute all the process is described in equation 1. Equation 2 gives the time required to automatically generate the same approach. The following list of terms describe, each action, in both equations (m being the number of tests, n the number of inputs and l the number of executed tests):

- T_MAN: the total time needed in the manual process.
- T_VF: the total time needed in the automatic process.
- T_MConf: the manual configuration of all the tests.
- T_MInp: the manual creation of all the inputs.
- T_CTest: the configuration of the tests that will be executed.
- T_ITest: the definition of the inputs for the tests that will be executed.
- T_SUT: the selection of a file with which the algorithm that Elevate will use to run the tests.
- T_COra: the configuration of the oracles.
- T_VStrat: the validation strategy that will be used for the executed tests.
- T_EleM: the manual execution of Elevate.
- T_EleA: the automatic execution of Elevate.
- T_MRes: the check of the evaluation results manually one by one.
- T_ARes: the check of the evaluation results automatically.

$$T\_MAN = \sum_{i=1}^{m} T\_MConf_i + \sum_{j=1}^{n} T\_MInp_j + \sum_{k=1}^{l} (T\_CTest_k + T\_ITest_k + T\_SUT_k + T\_EleM_k + T\_MRes_k) \quad (1)$$

$$T\_VF = \sum_{i=1}^{m} T\_MConf_i + \sum_{j=1}^{n} T\_MInp_j + T\_CTest + T\_ITest + T\_SUT + T\_COra + T\_VStrat + T\_EleA + T\_ARes \quad (2)$$

The difference between the element with sub-index k and the same elements without sub-index is that the action with sub-index is repeated k times and the others without sub-index, just once.

9.2. RQ1: Has the time needed to execute and evaluate the tests been reduced?

The manual approach is a very time-consuming procedure, and it was mandatory to reduce the time necessary in the multi-level testing process. The time needed in the step 1 in both approaches is exactly the same. The person who defines the tests and inputs needs a large number of minutes to define all of them because the person must be very precise defining them. The better the tests defined, the better the testing. The step 2 is different for the manual and for the validation workflow. In this step, 3 actions are the same (test selection, input selection and the evaluation algorithm selection) but one is executed manually and the other automatically. The manual process is time-consuming and a tester is on charge of selecting the better tests. Finally, the multi-level testing algorithm needs minutes to complete the task. The automatic approach includes 2 more steps (oracle and validation plan selection) but all are executed automatically, and the time needed to complete the procedure is measured in milliseconds. Similarly, the manual execution of step 3 is time-consuming because a person is required to configure Elevate and execute each test one by one, which increases the time to minutes. However, the automatic process executes the tests automatically, leading to an execution time in the range of milliseconds. The final step has also been considerably reduced in time. The manual mode demands the check of each result one by one, and this action is performed by a person that requires minutes to complete the task. The validation workflow checks the results automatically, and the time needed is measured in milliseconds.

Figure 12 shows graphically how since the beginning, the time needed in the process is more elevated in the manual mode than in the automatic mode. At the beginning, the time needed is almost the same, although the manual mode is more elevated. The difference between the two modes is more elevated when the number of tests increases. The more tests are executed, the more difference between the two modes is obtained. If 10 tests are executed, the execution time is reduced to a 56.91%.

9.3. RQ2: Has the economic cost needed to execute and evaluate the test been reduced?

It is important to remark that this multi-level testing process is performed every time a code change is implemented. This means that it is executed every day or more than once every week. This shows the importance of reducing the testing process and consequently, the economic impact of this time reduction is clear. It is noteworthy that the stakeholders who are involved are test engineers, whose cost is €30/hour.

Figure 13 shows the difference from the economic perspective of the automatic (machines) and manual modes (human). All approaches are linear. Nevertheless,

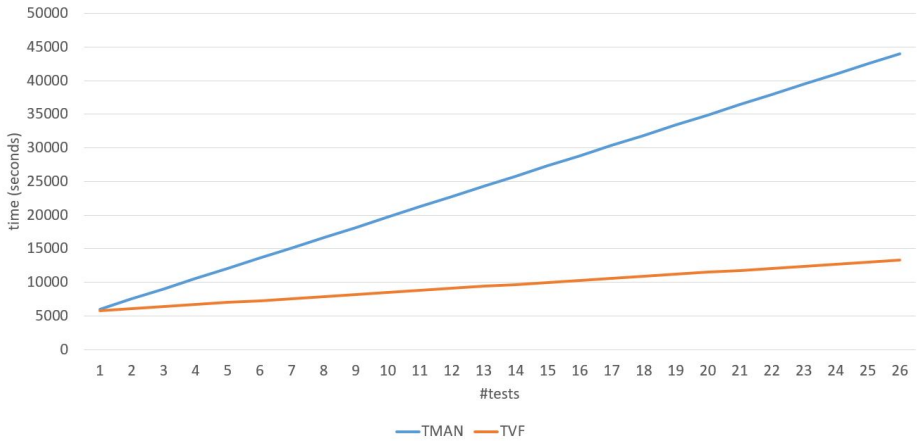

Figure 12. Comparison between the manual and automatic approaches

the automatic approaches are under the manual approach and the difference between the two (manual vs automatic) approaches is more elevated when the number of months increases. The equation used to calculate the economic cost have been extracted from the Bataev and Aleksandrova (2020) contribution. It has been utilized as follows (algorithm 3):

$$C = \frac{1}{1+r}(N_{red} * (S_{red} + T_{red}) - N_{add} * (S_{add} + T_{add})) \tag{3}$$

where:

- r is the number of machines
- N_{red} is the number of reduced jobs in the implementation of an industrial machine;
- N_{add} is the number of additional jobs that occur when implementing an industrial machine;
- S_{red} is the salary of one reduced person;
- T_{red} is taxes on the salary of one reduced person;
- S_{add} is the salary of one additional job created during the implementation of an industrial machine;
- T_{add} is taxes on the salary of one additional job created during the implementation of an industrial machine.

Figure 13 exhibits the differences between automatic and manual approaches, but also the difference between using one or more machines in parallel. The economic cost reduction between a human and a machine is 53.3% per month. Each month that our proposal is used, it will imply a very important cost reduction. In the case of using 4 machines, we would be talking about a cost reduction of 88.3% each month compared to the cost of one person. Therefore, we are talking about an important cost reduction. We must bear in mind that scaling our solution would have a minimal economic impact. Finally, we can see

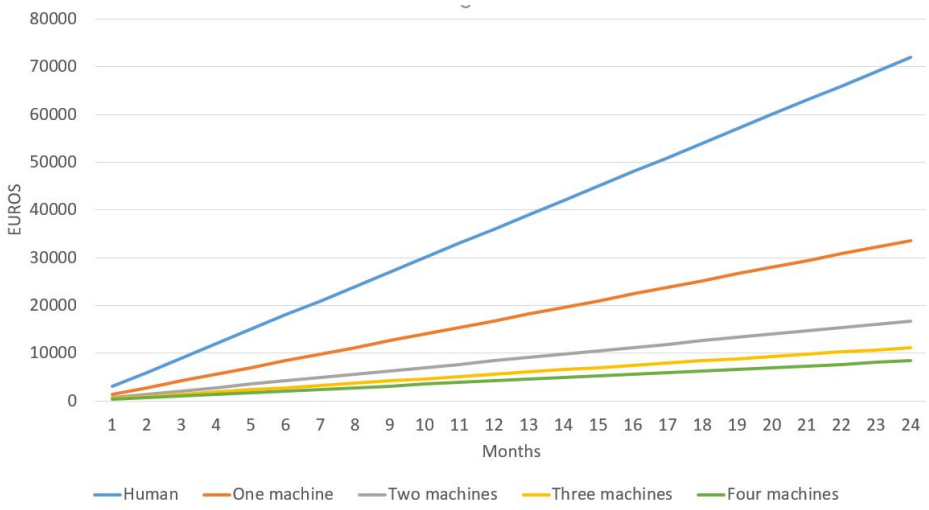

Figure 13. Economic comparison between the manual and automatic approaches

that the higher the number of machines, the cost reduction is not as significant. The reduction between one and two machines is 50% while the reduction between 3 and 4 machines is 25%.

9.4. Threats to validity

External validity: As for the execution time, the main threat to validity is related with the different execution time. Each execution needs a different period of time to be executed. For example, T_ITest execution time is not always the same. However, the intervals are not elevated and as a consequence we have used an average value to calculate the final execution time.

Regarding the economic savings, the cost of the workers is different, it depends on the worker status. However, differences are not important, and we have used an average salary for the evaluation. In addition, we have not calculated the optimum number of machines for the automatic process. Furthermore, based on Bataev and Aleksandrova (2020) contribution, we have not calculated the cost of implementing a CPS and the costs of the components of the production CPS. With this data, we would be able to calculate from what month the automatic mode compensated the cost of the automation process of the CPS.

10. Conclusion and future work

The use of microservices for CPSs has been extended in the last years. In this work, we report on the experience of instantiating of the microservice-based architecture for DevOps of CPSs (Gartziandia, 2021) to test elevator dispatching algorithms across different test levels (i.e., SiL, HiL and Operation). The first step has been the definition of a validation plan file based on the UTP2 standard,

in which all needed concepts for the multi-level testing process are described. Afterward, we have described generic Agile Testing Framework developed which is integrated in the DevOps ecosystem (Gartziandia, 2021). The framework implements two microservices for the validation orchestrator and validation agent. Furthermore, with the goal of demonstrating the feasibility and scalability of the approach with our industrial case study, we have explained the instantiation of the approach for the SiL test level of Orona's dispatching algorithms. Finally, an evaluation has been conducted to demonstrate that the consequences showed in the problem analysis, i.e., the huge quantity of time needed to execute test cases and the elevated economic cost for companies of developing tests for CPSs, are palliated. This evaluation demonstrates that our proposal is able to reduce considerably the time needed for the complete multi-level testing process and consequently, it reduces the economic cost of this process.

In the future, we would like to tackle some of the causes of the problem showed in the problem analysis. One of the causes is the lack of reusability and interoperability between test artifacts among the different test levels. We will define some common criteria for different types of test levels and, consequently, permit the interoperability between different test levels in different situations. Moreover, we desire to solve the difference between design-time methods and operation time methods. The reason is that these methods are usually performed by different designers and when a designer is expert in a field, they focus on the main aspects of their expertise. The consequence is that they neglect or avoid considering other important aspects, causing that some requirements are not completely fulfilled. The main goal would the improvement and generalization of our process that avoid this disadvantage.

Conflict of interest

The authors report no potential conflict of interest.

Data Availability Statement

Data sharing not applicable to this article as no datasets were generated or analyzed during the current study.

Authors Contribution

Aitor Arrieta, Maite Arratibel, Goiuria Sagardui and Aitor Agirre conceived of the presented idea. Iñigo Aldalur, Aitor Arrieta, Maite Arratibel and Aitor Agirre developed the theory and performed the computations. Iñigo Aldalur and Aitor Arrieta carried out the experiment. Iñigo Aldalur and Aitor Arrieta wrote the manuscript with support from Maite Arratibel, Goiuria Sagardui and Aitor Agirre

Funding Declaration

This publication is part of a project that has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 871319. Iñigo Aldalur, Aitor Arrieta and Goiuria Sagardui are part of the Software and Systems Engineering research group of Mondragon Unibertsitatea (IT519-22), supported by the Department of Education, Universities and Research of the Basque Government.

References

Armengaud, E., Macher, G., Massoner, A., Frager, S., Adler, R., Schneider, D., Longo, S., Melis, M., Groppo, R., Villa, F., *et al.*: 2018, Deis: dependability engineering innovation for industrial cps. In: *Advanced Microsystems for Automotive Applications 2017*, 151. Springer. [11]

Arrieta, A., Sagardui, G., Etxeberria, L., Zander, J.: 2017a, Automatic generation of test system instances for configurable cyber-physical systems. *Software Quality Journal* **25**(3), 1041. [arrieta2017automatic]

Arrieta, A., Wang, S., Markiegi, U., Sagardui, G., Etxeberria, L.: 2017b, Employing multi-objective search to enhance reactive test case generation and prioritization for testing industrial cyber-physical systems. *IEEE Transactions on Industrial Informatics* **14**(3), 1055. [67]

Arrieta, A., Wang, S., Markiegi, U., Sagardui, G., Etxeberria, L.: 2017c, Search-based test case generation for cyber-physical systems. In: *2017 IEEE Congress on Evolutionary Computation (CEC)*, 688. IEEE. [66]

Arrieta, A., Wang, S., Markiegi, U., Arruabarrena, A., Etxeberria, L., Sagardui, G.: 2019a, Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information and Software Technology* **114**, 137. [65]

Arrieta, A., Wang, S., Sagardui, G., Etxeberria, L.: 2019b, Search-based test case prioritization for simulation-based testing of cyber-physical system product lines. *Journal of Systems and Software* **149**, 1. [64]

Arrieta, A., Agirre, A., Sagardui, G., Arratibel, M.: 2022, Design-operation continuum methods for traffic master. In: *13th Symposium on Lift & Escalator Technologies* **13**, 41. [50]

Arrieta, A., Valle, P., Agirre, J.A., Sagardui, G.: 2023, Some seeds are strong: Seeding strategies for search-based test case selection. *ACM Transactions on Software Engineering and Methodology* **32**(1), 1. [63]

ASAMRef.: Asam xil. *https://www.asam.net/standards/detail/xil/*. Accessed: 2023-04-01. [68]

Avizienis, A., Laprie, J., Randell, B., Landwehr, C.E.: 2004, Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.* **1**(1), 11. [10]

Ayerdi, J., Segura, S., Arrieta, A., Sagardui, G., Arratibel, M.: 2020a, Qos-aware metamorphic testing: An elevation case study. In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 104. IEEE. [62]

Ayerdi, J., Garciandia, A., Arrieta, A., Afzal, W., Enoiu, E., Agirre, A., Sagardui, G., Arratibel, M., Sellin, O.: 2020b, Towards a taxonomy for eliciting design-operation continuum requirements of cyber-physical systems. In: *28th IEEE International Requirements Engineering Conference, RE 2020, Zurich, Switzerland, August 31 - September 4, 2020*, 280. IEEE. [3]

Ayerdi, J., Terragni, V., Arrieta, A., Tonella, P., Sagardui, G., Arratibel, M.: 2021, Generating metamorphic relations for cyber-physical systems with genetic programming: an industrial case study. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1264. [61]

Ayerdi, J., Valle, P., Segura, S., Arrieta, A., Sagardui, G., Arratibel, M.: 2022, Performance-driven metamorphic testing of cyber-physical systems. *IEEE Transactions on Reliability*. [60]

Bataev, A.V., Aleksandrova, A.: 2020, Digitalization of the world economy: Performance evaluation of introducing cyber-physical systems. In: *2020 9th International Conference on Industrial Technology and Management (ICITM)*, 265. IEEE. [20]

Birchler, C., Khatiri, S., Bosshard, B., Gambi, A., Panichella, S.: 2022, Machine learning-based test selection for simulation-based testing of self-driving cars software. *arXiv preprint arXiv:2212.04769*. [59]

Briand, L., Nejati, S., Sabetzadeh, M., Bianculli, D.: 2016, Testing the untestable: model testing of complex software-intensive systems. In: *Proceedings of the 38th international conference on software engineering companion*, 789. [58]

Bruyninckx, H.: 2008, Robotics software: The future should be open [position]. *IEEE Robotics & Automation Magazine* **15**(1), 9. [6]

Gartziandia, A.: 2021, Microservice-based performance problem detection in cyber-physical system software updates. In: *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021*, 147. IEEE. [25]

Gartziandia, A., Arrieta, A., Agirre, A., Sagardui, G., Arratibel, M.: 2021, Using regression learners to predict performance problems on software updates: a case study on elevators dispatching algorithms. In: *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*, 135. ACM. [16]

González-Nalda, P., Etxeberria-Agiriano, I., Calvo, I., Otero, M.C.: 2017, A modular cps architecture design based on ros and docker. *International Journal on Interactive Design and Manufacturing (IJIDeM)* **11**(4), 949. [7]

He, Z., Chen, Y., Huang, E., Wang, Q., Pei, Y., Yuan, H.: 2019, A system identification based oracle for control-cps software fault localization. In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 116. IEEE / ACM. [17]

Hu, T., Bertolotti, I.C., Navet, N., Havet, L.: 2020, Automated fault tolerance augmentation in model-driven engineering for CPS. *Comput. Stand. Interfaces* **70**, 103424. [5]

Humeniuk, D., Khomh, F., Antoniol, G.: 2022, A search-based framework for automatic generation of testing environments for cyber–physical systems. *Information and Software Technology* **149**, 106936. [57]

Khatiri, S., Panichella, S., Tonella, P.: 2023, Simulation-based test case generation for unmanned aerial vehicles in the neighborhood of real flights. In: *16th IEEE International Conference on Software Testing, Verification and Validation (ICST), Dublin, Ireland, 16-20 April 2023*. ZHAW Zürcher Hochschule für Angewandte Wissenschaften. [56]

Lampke, S., Schliecker, S., Ziegenbein, D., Hamann, A.: 2015, Resource-aware control-model-based co-engineering of control algorithms and real-time systems. *SAE International Journal of Passenger Cars-Electronic and Electrical Systems* **8**(2015-01-0168), 106. [14]

Lee, E.A., Seshia, S.A.: 2017, *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press. [1]

Masin, M., Palumbo, F., Myrhaug, H., de Oliveira Filho, J.A., Pastena, M., Pelcat, M., Raffo, L., Regazzoni, F., Sanchez, A.A., Toffetti, A., de la Torre, E., Zedda, K.: 2017, Cross-layer design of reconfigurable cyber-physical systems. In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, 740. IEEE. [4]

Matinnejad, R., Nejati, S., Briand, L.C., Bruckmann, T.: 2019, Test generation and test prioritization for simulink models with dynamic behavior. *IEEE Trans. Software Eng.* **45**(9), 919. [15]

Menghi, C., Nejati, S., Gaaloul, K., Briand, L.C.: 2019, Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. In: *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 27. [55]

Menghi, C., Nejati, S., Briand, L., Parache, Y.I.: 2020, Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 372. [54]

Miele, A.: 2014, A fault-injection methodology for the system-level dependability analysis of multiprocessor embedded systems. *Microprocess. Microsystems* **38**(6), 567. [12]

Neema, H., Sztipanovits, J., Steinbrink, C., Raub, T., Cornelsen, B., Lehnhoff, S.: 2019, Simulation integration platforms for cyber-physical systems. In: *Proceedings of the Workshop*

on Design Automation for CPS and IoT, DESTION@CPSIoTWeek 2019, Montreal, QC, Canada, April 15, 2019, 10. ACM. [8]

Nejati, S., Gaaloul, K., Menghi, C., Briand, L.C., Foster, S., Wolfe, D.: 2019, Evaluating model testing and model checking for finding requirements violations in simulink models. In: *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 1015. [53]

Perez, A.M., Kaiser, S.: 2009, Integrating test levels for embedded systems. In: *2009 Testing: Academic and Industrial Conference-Practice and Research Techniques*, 184. IEEE. [37]

Sadri-Moshkenani, Z., Bradley, J., Rothermel, G.: 2022, Survey on test case generation, selection and prioritization for cyber-physical systems. *Software Testing, Verification and Reliability* **32**(1), e1794. [52]

Sanislav, T., Mois, G., Miclea, L.: 2016, An approach to model dependability of cyber-physical systems. *Microprocessors and Microsystems* **41**, 67. [35]

Shin, S.Y., Nejati, S., Sabetzadeh, M., Briand, L.C., Zimmer, F.: 2018, Test case prioritization for acceptance testing of cyber physical systems: a multi-objective search-based approach. In: *Proceedings of the 27th acm sigsoft international symposium on software testing and analysis*, 49. [51]

Shin, S.Y., Chaouch, K., Nejati, S., Sabetzadeh, M., Briand, L.C., Zimmer, F.: 2021, Uncertainty-aware specification and analysis for hardware-in-the-loop testing of cyber-physical systems. *J. Syst. Softw.* **171**, 110813. [18]

Siikonen, M.: 1993, Elevator traffic simulation. *Simul.* **61**(4), 257. [31]

Stankovic, J.A.: 2016, Research directions for cyber physical systems in wireless and mobile healthcare. *ACM Trans. Cyber Phys. Syst.* **1**(1), 1:1. [2]

Sundharam, S.M., Navet, N., Altmeyer, S., Havet, L.: 2018, A model-driven co-design framework for fusing control and scheduling viewpoints. *Sensors* **18**(2), 628. [13]

UMLTestingProfile: 2018, Uml testing profile 2 (utp 2). *https://www.omg.org/spec/UTP2/2.0/PDF*. Accessed: 2022-11-29. [32]

Veledar, O., Damjanovic-Behrendt, V., Macher, G.: 2019, Digital twins for dependability improvement of autonomous driving. In: *Systems, Software and Services Process Improvement - 26th European Conference, EuroSPI 2019, Edinburgh, UK, September 18-20, 2019, Proceedings*, *Communications in Computer and Information Science* **1060**, 415. Springer. [9]

Wang, C., Pastore, F., Goknil, A., Briand, L.C., Iqbal, M.Z.Z.: 2015, Automatic generation of system test cases from use case specifications. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, 385. ACM. [19]

Zander-Nowicka, J., Schieferdecker, I., Perez, A.M.: 2006, Automotive validation functions for on-line test evaluation of hybrid real-time systems. In: *2006 IEEE Autotestcon*, 799. IEEE. [zander2006automotive]